# Report

## 1. System Selection and Setup

For single-class probe detection in Elios3 images, I chose YOLOv8-s from Ultralytics. It offers an excellent balance between precision, latency, and size, as well as production-ready tools: anchor-free head, strong data augmentations, easy transfer learning, and export to ONNX/TensorRT for Jetson deployment. Compared to newer variants (YOLOv9–v12), YOLOv8 has mature docs, stable APIs, and broad community adoption, which reduces integration risk and debugging time under a tight deadline. Given the goal (real-time detection on constrained hardware), YOLOv8-s is a pragmatic default: small footprint, fast inference, and competitive mAP.

YOLOv8 adopts an anchor-free head, eliminating the need for predefined anchor boxes and directly regressing bounding boxes. This reduces hyperparameter tuning, improves generalization, and is more robust for datasets with limited variation (such as a single probe class). Additionally, I selected the YOLOv8-small variant for its small footprint (lower parameter count and memory usage), which allows faster inference and better suitability for edge deployment on Jetson devices.

## 2. Training and Fine-Tuning

I started with 308 labeled probe images annotated in COCO JSON format. Since YOLO requires TXT-format labels, I developed a script (jsonToYOLO.py) to automatically convert the dataset. During the conversion, I split the dataset into 80% training, 10% validation, and 10% test sets (246/31/31 images) Notably, each image contains a probe, so the dataset has no true negatives, which impacts the evaluation metrics.

- I set 100 epochs as an upper bound, allowing sufficient iterations for convergence, while relying on early stopping to avoid overfitting.
- The batch size was set to auto mode, allowing YOLO to maximize usage of available GPU memory without manual configuration.
- Training was performed on GPU in Google Collab (device 0) for faster convergence.
- I set 4 data-loading workers so that images are prepared in parallel (resized, augmented, etc.)
- I enabled early stopping with patience=20, ensuring training stops once the model no longer improves on validation data.
- A fixed seed (123) was set to guarantee reproducibility of dataset splits and training results.

## 3. Evaluation

### Detection Performance

The trained YOLOv8-small model was evaluated on both validation and test sets. On the validation set, the system achieved Precision = 0.997, Recall = 0.968, and F1-score = 0.982 at IoU = 0.5. The mean Average Precision was mAP@50 = 0.991 and mAP@50–95 = 0.909. On the test set, results were very similar, with mAP@50 = 0.995 and mAP@50–95 = 0.922, confirming strong generalization.

### Inference Runtime

Runtime was measured on a local PC using only the CPU. The model processed 31 test images with an average time of 123.4 ms per image, which corresponds to about 8.1 FPS. This timing includes the full pipeline (image loading, preprocessing, model inference, postprocessing, and visualization). While this is below real-time video speed, it is expected on CPU. On GPU or with deployment optimizations such as TensorRT FP16/INT8, throughput can be significantly higher and closer to real-time performance.

### Analysis

**Strengths:**

- Very high precision and recall, showing the model is reliable in detecting probes.
- Stable performance between validation and test sets, which indicates no overfitting.
- Tight bounding boxes confirmed by high IoU-based scores.

**Weaknesses:**

- Dataset only contains images with probes, meaning real-world false positives on probe-like objects might be underestimated.
- Performance on more challenging cases (e.g., occluded or blurred probes) cannot be fully assessed with the current dataset.

**Trends:**

- Training converged quickly (within ~40 epochs) and metrics stabilized at high values.

## 4. Future improvements.

### 1) Improve system's performance or robustness

- **Add true negatives & hard negatives**: Include images *without* probes.
- **Resolution & model sweep**: Compare `imgsz=512/640/768` and `YOLOv8n/s`. Pick the best trade-off between recall and speed for the device selected.
- **Active learning loop**: After first deployment, collect the model's mistakes (FP/FN), label them, and retrain. Fast way to boost real-world performance.
- **Baseline comparison**: Due to time constraints, I did not run these baselines here, but they would serve as useful points of comparison in future iterations. I would consider **classical methods** as template matching or color-based blob detection as simple models to compare with.

### 2) Inference runtime improvements

- **Export to TensorRT**: Convert the model to a TensorRT engine and run in FP16 (and later INT8) for big speed gains.
- **Use a smaller variant**: If needed, switch to **YOLOv8n** (nano).
- **Right-size the input**: Test **imgsz=512 or 576**; often close to the same accuracy as 640 but faster.

So the main steps are:

1. Add **~100–300** negative/hard-negative frames and retrain.
2. Export to **TensorRT** and test on Jetson. If needed, switch to **YOLOv8n**.